Module 1: The AI-Augmented Environment, Lesson 1: The Accelerated Mindset

Hands-On: Oracle vs. Execution Engine—The Constraint Experiment

Overview

This demo proves three claims from the lecture in real time:

- An LLM given a vague chat-style prompt will fill the gaps with its own assumptions—and silently break your architecture
- The same task, framed with explicit ground truth and constraints, produces predictable, auditable output
- Constraint management—not syntax—is the new high-leverage engineering skill

You will run the same engineering task through your AI tool three times in one session: first as a conversational prompt, then as a directed Execution Engine prompt, then as an Architect's Audit of the result. You will experience the difference between treating the AI as an oracle and treating it as an execution engine that depends entirely on the constraints you provide.

Before You Start

- Open your AI tool of choice: Claude.ai, ChatGPT, Cursor, or GitHub Copilot Chat
- Open a blank scratch document or notes app to record observations after each prompt
- Keep two windows visible side by side if possible: AI tool on the left, this handout on the right
- Do NOT start a new chat between prompts—session continuity is part of the lesson

> **Remember**
>
> LLMs are not oracles. They do not hold ground truth. They predict likely completions based on the context you provide. If you leave gaps, the model fills them with assumptions—and those assumptions are where your architecture breaks. In the next minutes, you are going to watch this happen in real time, then fix it.

During Hands-On Keep Visible if Possible

1. **ROLE** — who is the AI acting as (senior engineer? security reviewer?)
2. **GROUND TRUTH** — stack, version, storage, schema, existing constraints
3. **INTENT** — what the code must do, in one sentence
4. **CONSTRAINTS** — what is NOT allowed, limits, SLAs, dependencies to avoid
5. **OUTPUT FORMAT**— exact shape of the deliverable (function signature, no tests, etc.)

Prompt 1 — The Chat Trap (Vague Prompt)

> **What to Learn**
>
> This is how most engineers prompt today—one casual sentence, no context. Watch what the model invents to fill the silence. Every invention is an assumption you did not make, and every assumption is a future bug waiting to surface in staging.

Paste this into the AI chat exactly as written. Do not add context:

```
Write me a Python function to rate-limit API requests.
```

> **What you should get**
>
> A plausible-looking function—probably a decorator using an in-memory dict or the time module, hardcoded limits like 100 requests per minute, and no mention of your framework, storage, or auth. The AI has invented: the framework, the storage backend, the rate (100? 60? per what?), the scope (per user? per IP? global?), the response when the limit is hit, and whether this runs in a single process or a distributed system. None of that came from you.

Record in your notes:

- How many design decisions did the AI make without asking you?
- Would this code survive behind a load balancer with 4 app servers? (Hint: check the storage backend)
- Would it pass your team's code review today?

You should ask:

“Where did the AI get the number 100? Where did it get the storage backend? Where did it get the scope?”

Answer: nowhere. It predicted a likely completion based on training data averages. That is exactly what the lecture meant by *“treating the LLM as an oracle.”* The output looks confident because LLMs always sound confident. That is the trap.

Prompt 2 — Directing the Execution Engine

> **Remember**
>
> Same chat window. The AI already has the previous vague output in context. We are now going to establish ground truth and explicit constraints—the five blocks on your visible checklist. Watch the output collapse from “generic plausible code” to “code that matches your stack.”

Continue in the same chat session—do NOT start a new one. Paste this:

```
Redo the rate-limiter using these explicit inputs.

ROLE: You are a senior backend engineer reviewing a PR for production.

GROUND TRUTH:
- Stack: FastAPI 0.110 on Python 3.11, running in 4 containerized
replicas
  behind an AWS ALB
- Shared state store: Redis 7 (already provisioned, client is
redis.asyncio)
- Auth: requests carry a verified JWT with a `sub` claim (user id)
- Observability: structlog is the team logger; never use print()

INTENT: Enforce a per-user rate limit on a FastAPI dependency so that any
route using the dependency returns 429 when the user exceeds the limit.

CONSTRAINTS:
- Must work correctly across all 4 replicas (no in-process dicts, no
lru_cache)
- Limit: 60 requests per 60 seconds per user, sliding window
- On limit hit: return 429 with a Retry-After header in seconds
- No new dependencies beyond redis, fastapi, structlog
- Never block the event loop (all Redis calls must be awaited)
- Do NOT write tests, do NOT write the FastAPI app wiring

OUTPUT FORMAT:
```

```
- One file: rate_limit.py
- One async function `rate_limit_dependency(request) -> None` suitable
for
  use with `Depends(...)`
- Type hints on every parameter and return
- A docstring on the dependency explaining the algorithm in 3 lines
```

> **What you should get**
>
> A single file with a FastAPI async dependency that uses Redis (not an in-memory dict), extracts the user id from the JWT claim you specified, applies a sliding-window algorithm, returns HTTP 429 with a Retry-After header, and logs through structlog. Every decision the AI made in Prompt 1 has been reversed—because you, not the model, supplied the ground truth.

Side-by-side comparison—record in your notes:

- Count assumptions. Prompt 1: how many? Prompt 2: how many?
- Multi-replica safety: does Prompt 1 work behind 4 containers? Does Prompt 2?
- Scope: per user? per IP? global? Which prompt forced the answer to be deterministic?

Tip: *If the Prompt 2 output still drifted (used the wrong client, invented a library, skipped the header), that is not AI failure. That is a gap in your CONSTRAINTS block. Tighten the prompt, do not retry blindly. This is constraint management.*

Prompt 3 — The Architect's Audit

> **What to Learn**
>
> Speed does not eliminate the need for review. Treat the Prompt 2 output like a pull request from a brilliant but inexperienced junior developer. They can write syntax; they do not know your business logic, your threat model, or your SLA. You do. This final prompt makes the AI argue against its own code so you can see where it is thin.

Still in the same session—paste this:

```
Stop generating code. Switch roles to senior reviewer.

Review the rate_limit.py you just produced and answer, with evidence
from the code you wrote:

1. CORRECTNESS: Under what race condition could two concurrent requests
   both be admitted when they should not be? Quote the lines.
2. SCALABILITY: If Redis latency spikes to 300ms, what happens to p99
   on this endpoint? What is the blast radius?
3. SECURITY: What happens if the JWT is missing the `sub` claim, or if
   `sub` is an attacker-controlled string? Is there an injection risk
   in any Redis key construction?
4. OBSERVABILITY: When this dependency rate-limits a user at 2am, what
   exactly shows up in the logs? Is it enough to diagnose? If not, what
   log fields are missing?

For each finding, cite the specific line, rate the risk (low/med/high),
and propose the minimal fix. Do not rewrite the whole file.
```

> **What you should get**
>
> Four cited findings. Expect at least one real issue—commonly a check-then-set race condition in the sliding window, a missing null check on the JWT claim, or logs that omit the user id and the limit value. This is the “architect-level questions only you can answer” pattern from the lecture. The AI wrote the code in seconds; you own correctness, scalability, and security.

Record in your notes:

- Which finding would you have caught in your own review? Which did the AI surface that you might have missed?
- Is there a finding the AI invented that is not actually a risk in your system? (This is why you audit, not merge.)

Wrap-Up — Mindset Shift, Locked In

> **Remember**
>
> Three prompts. One session. You made the same AI produce three entirely different artifacts—fragile code, production code, and a security review—by changing one thing: the

> constraints you supplied. The model did not get smarter between prompts. You got more precise. That is the entire thesis of this module.

Key takeaways:

- An LLM without constraints is an oracle guess. An LLM with constraints is an execution engine.
- Ground truth (stack, version, storage, auth) is not optional context—it is the foundation. Omit it and the model invents it.
- Your new high-leverage skill is Constraint Management. You define what gets foregrounded; the AI executes inside those walls.
- Speed without review is how technical debt compounds. The Architect's Audit is not a final step—it is the whole point.
- Next lesson (1B): we retrofit your IDE and privacy settings so these prompts run in a secure, zero-retention environment.